Clock-SI: Snapshot Isolation for Partitioned Data Stores Using Loosely Synchronized Clocks

Jiaqing Du
EPFL
Lausanne, Switzerland

Sameh Elnikety
Microsoft Research
Redmond, WA, USA

Willy Zwaenepoel
EPFL
Lausanne, Switzerland

***Abstract*—Clock-SI is a fully distributed protocol that implements snapshot isolation (SI) for partitioned data stores. It derives snapshot and commit timestamps from loosely synchronized clocks, rather than from a centralized timestamp authority as used in current systems. A transaction obtains its snapshot timestamp by reading the clock at its originating partition and Clock-SI provides the corresponding consistent snapshot across all the partitions. In contrast to using a centralized timestamp authority, Clock-SI has availability and performance benefits: It avoids a single point of failure and a potential performance bottleneck, and improves transaction latency and throughput.**

We develop an analytical model to study the trade-offs introduced by Clock-SI among snapshot age, delay probabilities of transactions, and abort rates of update transactions. We verify the model predictions using a system implementation. Furthermore, we demonstrate the performance benefits of Clock-SI experimentally using a micro-benchmark and an application-level benchmark on a partitioned key-value store. For short read-only transactions, Clock-SI improves latency and throughput by 50% by avoiding communications with a centralized timestamp authority. With a geographically partitioned data store, Clock-SI reduces transaction latency by more than 100 milliseconds. Moreover, the performance benefits of Clock-SI come with higher availability.

***Keywords*-snapshot isolation, distributed transactions, partitioned data, loosely synchronized clocks**

I. Introduction

Snapshot isolation (SI) [1] is one of the most widely used concurrency control schemes. While allowing some anomalies not possible with serializability [2], SI has significant performance advantages. In particular, SI never aborts read-only transactions, and read-only transactions do not block update transactions. SI is supported in several commercial systems, such as Microsoft SQL Server, Oracle RDBMS, and Google Percolator [3], as well as in many research prototypes [4], [5], [6], [7], [8].

Intuitively, under SI, a transaction takes a snapshot of the database when it starts. A snapshot is equivalent to a logical copy of the database including all committed updates. When an update transaction commits, its updates are applied atomically and a new snapshot is created. Snapshots are totally ordered according to their creation order using monotonically increasing timestamps. Snapshots are identified by timestamps: The snapshot taken by a transaction is identified by the transaction's *snapshot timestamp*. A new snapshot created by a committed update transaction is identified by the transaction's *commit timestamp*.

Managing timestamps in a centralized system is straightforward. Most SI implementations maintain a global variable, the *database version*, to assign snapshot and commit timestamps to transactions.

When a transaction starts, its snapshot timestamp is set to the current value of the database version. All its reads are satisfied from the corresponding snapshot. To support snapshots, multiple versions of each data item are kept, each tagged with a version number equal to the commit timestamp of the transaction that creates the version. The transaction reads the version with the largest version number smaller than its snapshot timestamp. If the transaction is read-only, it always commits without further checks. If the transaction has updates, its writes are buffered in a workspace. When the update transaction requests to commit, a certification check verifies that the transaction writeset does not intersect with the writesets of concurrent committed transactions. If the certification succeeds, the database version is incremented, and the transaction commit timestamp is set to this value. The transaction's updates are made durable and visible, creating a new version of each updated data item with a version number equal to the commit timestamp.

Efficiently maintaining and accessing timestamps in a distributed system is challenging. We focus here on partitioning, which is the primary technique employed to manage large data sets. Besides allowing for larger data sizes, partitioned systems improve latency and throughput by allowing concurrent access to data in different partitions. With current large main memory sizes, partitioning also makes it possible to keep all data in memory, further improving performance.

Existing implementations of SI for a partitioned data store [3], [5], [9], [4] use a centralized authority to manage timestamps. When a transaction starts, it requests a snapshot timestamp from the centralized authority. Similarly, when a successfully certified update transaction commits, it requests a commit timestamp from the centralized authority. Each partition does its own certification for update transactions, and a two-phase commit (2PC) protocol is used to commit transactions that update data items at multiple partitions. The centralized timestamp authority is a single point of failure and a potential performance bottleneck. It negatively impacts system availability, and increases transaction latency and messaging overhead. We refer to the implementations of SI using a centralized timestamp authority as *conventional SI*.

This paper introduces Clock-SI, a fully distributed implementation of SI for partitioned data stores. Clock-SI uses loosely synchronized clocks to assign snapshot and commit timestamps to transactions, avoiding the centralized timestamp authority in conventional SI. Similar to conventional SI, partitions do their own certification, and a 2PC protocol is used to commit transactions that update multiple partitions.

Compared with conventional SI, Clock-SI improves system availability and performance. Clock-SI does not have a single point of failure and a potential performance bottleneck. It saves one round-trip message for a ready-only transaction (to obtain the snapshot timestamp), and two round-trip messages for an update transaction (to obtain the snapshot timestamp and the commit timestamp). These benefits are significant when the workload consists of short transactions as in key-value stores, and even more prominent when the data set is partitioned geographically across data centers.

We build on earlier work [10], [11], [12] to totally order events using physical clocks in distributed systems. The novelty of Clock-SI is to efficiently create consistent snapshots using loosely synchronized clocks. In particular, a transaction's snapshot timestamp is the value of the local clock at the partition where it starts. Similarly, the commit timestamp of a local update transaction is obtained by reading the local clock.

The implementation of Clock-SI poses several challenges because of using loosely synchronized clocks. The core of these challenges is that, due to a clock skew or pending commit, a transaction may receive a snapshot timestamp for which the corresponding snapshot is not yet fully available. We delay operations that access the unavailable part of a snapshot until it becomes available. As an optimization, we can assign to a transaction a snapshot timestamp that is slightly smaller than the clock value to reduce the possibility of delayed operations.

We build an analytical model to study the properties of Clock-SI and analyze the trade-offs of using old snapshots. We also verify the model using a system implementation.

We demonstrate the performance benefits of Clock-SI on a partitioned key-value store using a micro-benchmark (YCSB [13]) and application-level benchmark (Twitter feed-following [14]). We show that Clock-SI has significant performance advantages. In particular, for short read-only transactions, Clock-SI improves latency and throughput by up to 50% over conventional SI. This performance improvement comes with higher availability as well.

In this paper, we make the following contributions:

- We present Clock-SI, a fully distributed protocol that implements SI for partitioned data stores using loosely synchronized clocks (Section III).
- We develop an analytical model to study the performance properties and trade-offs of Clock-SI (Section IV).
- We build a partitioned key-value store and experimentally evaluate Clock-SI to demonstrate its performance benefits (Section V).

II. Background and Overview

In this section, we define the system model and SI, and describe the challenges of using loosely synchronized physical clocks to implement SI.

A. System Model

We consider a multiversion key-value store, in which the dataset is partitioned and each partition resides on a single server. A server has a standard hardware clock. Clocks are synchronized by a clock synchronization protocol, such as Network Time Protocol (NTP) [15]. We assume that clocks always move forward, perhaps at different speeds as provided by common clock synchronization protocols [15], [16]. The absolute value of the difference between clocks on different servers is bounded by the clock synchronization skew.

The key-value store supports three basic operations: `get`, `put`, and `delete`. A transaction consists of a sequence of basic operations. A client connects to a partition, selected by a load balancing scheme, and issues transactions to the partition. We call this partition the *originating partition* of transactions from the connected client. The originating partition executes the operations of a transaction sequentially. If the originating partition does not store a data item needed by an operation, it executes the operation at the remote partition that stores the item.

The originating partition assigns the snapshot timestamp to a transaction by reading its local clock. When an update transaction starts to commit, if it updates data items at a single partition, the commit timestamp is assigned by reading the local clock at that partition. We use a more complex protocol to commit a transaction that updates multiple partitions (see Section III-B).

B. Snapshot Isolation

Formally, SI is a multiversion concurrency control scheme with three main properties [1], [17], [18] that must be satisfied by the underlying implementation: (1) Each transaction reads from a consistent snapshot, taken at the start of the transaction and identified by a snapshot timestamp. A snapshot is consistent if it includes all writes of transactions committed before the snapshot timestamp, and if it does not include any writes of aborted transactions or transactions committed after the snapshot timestamp. (2) Update transactions commit in a total order. Every commit produces a new database snapshot, identified by the commit timestamp. (3) An update transaction aborts if it introduces a write-write conflict with a concurrent committed transaction. Transaction T_1 is concurrent with committed update transaction T_2, if T_1 took its snapshot before T_2 committed and T_1 tries to commit after T_2 committed.

C. Challenges

From the SI definition, a consistent snapshot with snapshot timestamp t includes, for each data item, the version written by the transaction with the greatest commit timestamp smaller than t. This property holds independent of where a transaction starts and gets its snapshot timestamp, where an update

transaction gets its commit timestamp, and where the accessed data items reside. Ensuring this property is challenging when assigning snapshot and commit timestamps using clocks as we illustrate here. While these situations happen relatively rarely, they must be handled for correctness. We show in detail how Clock-SI addresses these challenges in Section III.

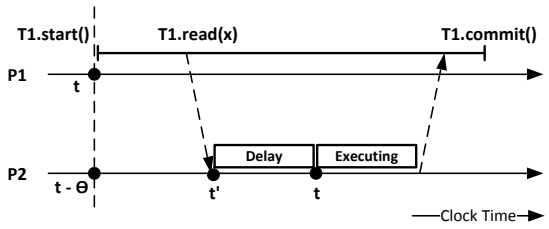

Fig. 1. Snapshot unavailability due to clock skew.

Example 1: First, we show that clock skew may cause a snapshot to be unavailable. Figure 1 shows a transaction accessing two partitions. Transaction T_1 starts at partition P_1, the originating partition. P_1 assigns T_1's snapshot timestamp to the value t. The clock at P_2 is behind by some amount θ, and thus at time t on P_1, P_2's clock value is $t-\theta$. Later on, T_1 issues a read for data item x stored at partition P_2. The read arrives at time t' on P_2's clock, before P_2's clock has reached the value t, and thus $t' < t$. The snapshot with timestamp t at P_2 is therefore not yet available. Another transaction on P_2 could commit at time t'', between t' and t, and change the value of x. This new value should be included in T_1's snapshot.

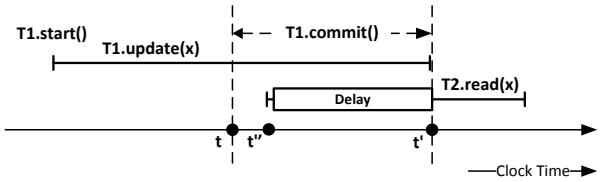

Fig. 2. Snapshot unavailability due to the pending commit of an update transaction.

Example 2: Second, we show that the pending commit of an update transaction can cause a snapshot to be unavailable. Figure 2 depicts two transactions running in a single partition. T_2's snapshot is unavailable due to the commit in progress of transaction T_1, which is assigned the value of the local clock, say t, as its commit timestamp. T_1 updates item x and commits. The commit operation involves a write to stable storage and completes at time t'. Transaction T_2 starts between t and t', and gets assigned a snapshot timestamp t'', $t < t'' < t'$. If T_2 issues a read for item x, we cannot return the value written by T_1, because we do not yet know if the commit will succeed, but we can also not return the earlier value, because, if T_1's commit succeeds, this older value will not be part of a consistent snapshot at t''.

Both examples are instances of a situation where the snapshot specified by the snapshot timestamp of a transaction is not yet available. These situations arise because of using physical clocks at each partition to assign snapshot and commit timestamps in a distributed fashion. We deal with these situations by delaying the operation until the snapshot becomes available.

As an optimization, the originating partition can assign to a transaction a snapshot timestamp that is slightly smaller than its clock value, with the goal of reducing the probability and duration that an operation needs to be delayed, albeit at the cost of reading slightly stale data. Returning to Example 1, if we assign a snapshot timestamp by subtracting the expected clock skew from the local clock, then the probability of the snapshot not being available because of clock skew decreases substantially.

III. Clock-SI

In this section, we describe how Clock-SI works. We first present the read protocol, which provides transactions consistent snapshots across partitions, and the commit protocol. Next, we discuss correctness and other properties of Clock-SI.

Algorithm 1 Clock-SI read protocol.

```
1:  StartTransaction(transaction T)
2:      T.SnapshotTime ← GetClockTime() − Δ
3:      T.State ← active
4:  ReadDataItem(transaction T, data item oid)
5:      if oid ∈ T.WriteSet return T.WriteSet[oid]
6:      // check if delay needed due to pending commit
7:      if oid is updated by T′ ∧
8:          T′.State = committing ∧
9:          T.SnapshotTime > T′.CommitTime
10:     then wait until T′.State = committed
11:     if oid is updated by T′ ∧
12:         T′.State = prepared ∧
13:         T.SnapshotTime > T′.PrepareTime ∧
14:         // Here T can obtain commit timestamp of T′
15:         // from its originating partition by a RPC.
16:         T.SnapshotTime > T′.CommitTime
17:     then wait until T′.State = committed
18:     return latest version of oid created before T.SnapshotTime
19: upon transaction T arriving from a remote partition
20:     // check if delay needed due to clock skew
21:     if T.SnapshotTime > GetClockTime()
22:     then wait until T.SnapshotTime < GetClockTime()
```

A. Read Protocol

The read protocol of Clock-SI provides transactions consistent snapshots across multiple partitions. It has two important aspects: (1) the assignment of snapshot timestamps, and (2) delaying reads under certain conditions to guarantee that transactions access consistent snapshots identified by their snapshot timestamps. Algorithm 1 presents the pseudocode of the read protocol.

Timestamp assignment. When transaction T is initialized at its originating partition (lines 1-3), it receives the snapshot timestamp by reading the local physical clock, and possibly subtracting a parameter, Δ, to access an older snapshot as we explain in Section III-C. The assigned timestamp determines the snapshot of the transaction.

Consistent Snapshot Reads. A transaction reads a data item by its identifier denoted by oid (lines 4-18). To guarantee that a transaction reads from a consistent snapshot, Clock-SI

delays a read operation until the required snapshot becomes available in two cases.

Case 1: Snapshot unavailability due to pending commit. Transaction T tries to access an item that is updated by another transaction T' which has a commit timestamp smaller than T's snapshot timestamp but has not yet completed the commit. For example, T' is being committed locally but has not completely committed (lines 6-10) or T' is prepared in 2PC (lines 11-17). [1] We delay T's access to ensure that a snapshot includes only committed updates and all the updates committed before the snapshot timestamp. The delay is bounded by the time of synchronously writing the update transaction's commit record to stable storage, plus one round-trip network latency in the case that a transaction updates multiple partitions.

Case 2: Snapshot unavailability due to clock skew. When a transaction tries to access a data item on a remote partition and its snapshot timestamp is greater than the clock time at the remote partition, Clock-SI delays the transaction until the clock at the remote partition catches up (lines 19-22). The transaction, therefore, does not miss any committed changes included in its snapshot. The delay is bounded by the maximum clock skew allowed by the clock synchronization protocol minus one-way network latency.

In both cases, delaying a read operation does not introduce deadlocks: An operation waits only for a finite time, until a commit operation completes, or a clock catches up.

Notice that Clock-SI also delays an update request from a remote partition, under the same condition that it delays a read request, so that the commit timestamp of an update transaction is always greater than the snapshot timestamp (line 19).

B. Commit Protocol

With Clock-SI, a read-only transaction reads from its snapshot and commits without further checks, even if the transaction reads from multiple partitions. An update transaction modifies items in its workspace. If the update transaction modifies a single partition, it commits locally at that partition. Otherwise, we use a coordinator to either commit or abort the update transaction at the updated partitions. One important aspect is how to assign a commit timestamp to update transactions. Algorithm 2 presents the pseudocode of the commit protocol.

Committing a single-partition update transaction. If a transaction updates only one partition, it commits locally at the updated partition (lines 5-10). Clock-SI first certifies the transaction by checking its writeset with concurrent committed transactions [18]. Before assigning the commit timestamp, the transaction state changes from *active* to *committing*. The updated partition reads its clock to determine the commit timestamp, and writes the commit record to stable storage.

[1]The question arises how T knows that it is reading a data item that has been written by a transaction in the process of committing, since in SI a write is not visible outside a transaction until it is committed. The problem is easily solved by creating, after assignment of a prepare or a commit timestamp, a version of the data item with that timestamp as its version number, but by prohibiting any transaction from reading that version until the transaction is fully committed.

Algorithm 2 Clock-SI commit protocol.

```
1:  CommitTransaction(transaction T)
2:      if T updates a single partition
3:      then LocalCommit(T)
4:      else DistributedCommit(T)
5:  LocalCommit(transaction T)
6:      if CertificationCheck(T) is successful
7:          T.State ← committing
8:          T.CommitTime ← GetClockTime()
9:          log T.CommitTime and T.Writeset
10:         T.State ← committed
11: // two-phase commit
12: DistributedCommit(transaction T)
13:     for p in T.UpdatedPartitions
14:         send prepare T to p
15:     wait until receiving T prepared from participants
16:     T.State ← committing
17:     // choose transaction commit time
18:     T.CommitTime ← max(all prepare timestamps)
19:     log T.CommitTime and commit decision
20:     T.State ← committed
21:     for p in T.UpdatedPartitions
22:         send commit T to p
23: upon receiving message prepare T
24:     if CertificationCheck(T) is successful
25:         log T.WriteSet and T's coordinator ID
26:         T.State ← prepared
27:         T.PrepareTime ← GetClockTime()
28:         send T prepared to T's coordinator
29: upon receiving message commit T
30:     log T.CommitTime
31:     T.State ← committed
```

Then, the transaction state changes from *committing* to *committed*, and its effects are visible in snapshots taken after the commit timestamp.

Committing a distributed update transaction. A multi-partition transaction, which updates two or more partitions, commits using an augmented 2PC protocol (lines 11-31) [2].

The transaction coordinator runs at the originating partition. Certification is performed locally at each partition that executed updates for the transaction (line 24). Each participant writes its prepare record to stable storage, changes its state from *active* to *prepared*, obtains the prepare timestamp from its local clock, sends the prepare message with the prepare timestamp to the coordinator, and waits for the response (line 25-28). The 2PC coordinator computes the commit timestamp as the maximum prepare timestamp of all participants (line 18).

Choosing the maximum of all prepare timestamps as the commit timestamp for a distributed update transaction is important for correctness. Remember from the read protocol that, on a participant, reads from transactions with a snapshot timestamp greater than the prepare timestamp of the committing transaction are delayed. If the coordinator were to return a commit timestamp smaller than the prepare timestamp on any of the participants, then a read of a transaction with a snapshot timestamp smaller than the prepare timestamp but greater than that commit timestamp would not have been delayed and

would have read an incorrect version (i.e., a version other than the one created by the committing transaction). Correctness is still maintained if a participant receives a commit timestamp greater than its current clock value. The effects of the update transaction will be visible only to transactions with snapshot timestamps greater than its commit timestamp.

C. Choosing Older Snapshots

In Clock-SI, the snapshot timestamp of a transaction is not restricted to the current value of the physical clock. We can choose the snapshot timestamp to be smaller than the clock value by Δ, as shown on line 2 of Algorithm 1. We can choose Δ to be any non-negative value and make this choice on a per-transaction basis. If we want a transaction to read fresh data, we set Δ to 0. If we want to reduce the delay probability of transactions close to zero, we choose an older snapshot by setting Δ to the maximum of (1) the time required to commit a transaction to stable storage synchronously plus one round-trip network latency, and (2) the maximum clock skew minus one-way network latency between two partitions. These two values can be measured and distributed to all partitions periodically. Since networks and storage devices are asynchronous, such a choice of the snapshot age does not completely prevent the delay of transactions, but it significantly reduces the probability.

While substantially reducing the delay probability of transactions, taking a slightly older snapshot comes at a cost: The transaction observes slightly stale data, and the transaction abort rate increases by a small fraction. We study this trade-off using an analytical model in Section IV and experiments on a prototype system in Section V.

D. Correctness

We show that Clock-SI implements SI by satisfying the three properties that define SI [18]. Furthermore, we show that safety is always maintained, regardless of clock synchronization precision.

(1) *Transactions commit in a total order.* Clock-SI assigns commit timestamps to update transactions by reading values from physical clocks. Ties are resolved based on partition ids. The commit timestamp order produces a total order on transaction commits.

(2) *Transactions read consistent snapshots.* The snapshot in Clock-SI is consistent with respect to the total commit order of update transactions. The snapshot timestamp specifies a snapshot from the totally ordered commit history. A transaction reads all committed changes of transactions with a smaller commit timestamp. By delaying transaction operations in the read protocol, a transaction never misses the version of a data item it is supposed to read. A transaction does not read values from an aborted transaction or from a transaction that commits with a greater commit timestamp.

(3) *Committed concurrent transactions do not have write-write conflicts.* Clock-SI identifies concurrent transactions by checking whether their execution time overlaps using the snapshot and commit timestamps. Clock skews do not affect the identification of concurrent transactions according to their snapshot and commit timestamps. Clock-SI aborts one of the two concurrent transactions with write-write conflicts.

We also point out an important property of Clock-SI: The precision of the clock synchronization protocol does not affect the correctness of Clock-SI but only the performance. Large clock skews increase a transaction's delay probability and duration; safety is, however, always maintained, satisfying the three properties of SI.

Although it does not affect Clock-SI's correctness, the transaction commit order in Clock-SI may be different from the real time commit order (according to global time) because of clock skews. This only happens to independent transactions at different partitions whose commit timestamp difference is less than the clock synchronization precision. Notice that this phenomenon also happens with conventional SI: A transaction commits on a partition after it obtains the commit timestamp from the timestamp authority. The asynchronous messages signaling the commit may arrive at the partitions in a different order from the order specified by the timestamps. Clock-SI preserves the commit order of dependent transactions when this dependency is expressed through the database as performed in a centralized system [2].

E. Discussion

Clock-SI is a fully distributed protocol. Compared with conventional SI, Clock-SI provides better availability and scalability. In addition, it also reduces transaction latency and messaging overhead.

Availability. Conventional SI maintains timestamps using a centralized service, which is a single point of failure. Although the timestamp service can be replicated to tolerate certain number of replica failures, replication comes with performance costs. In contrast, Clock-SI does not include such a single point of failure in the system. The failure of a data partition only affects transactions accessing that partition. Other partitions are still available.

Communication cost. With conventional SI, a transaction needs one round of messages to obtain the snapshot timestamp. An update transaction needs another round of messages to obtain the commit timestamp. In contrast, under Clock-SI, a transaction obtains the snapshot and commit timestamps by reading local physical clocks. As a result, Clock-SI reduces the latency of read-only transactions by one round trip and the latency of update transactions by two round trips. By sending and receiving fewer messages to start and commit a transaction, Clock-SI also reduces the cost of transaction execution.

Scalability. Since Clock-SI is a fully distributed protocol, the throughput of single-partition transactions increases as more partitions are added. In contrast, conventional SI uses a centralized timestamp authority, which can limit system throughput as it is on the critical path of transaction execution.

Session consistency. Session consistency [19], [20] guarantees that, in a workflow of transactions in a client session, each transaction sees (1) the updates of earlier committed

transactions in the same session, and (2) non-decreasing snapshots of the data. Session consistency can be supported under Clock-SI using the following standard approach. When a client finishes a read-only transaction, its snapshot timestamp is returned. When a client successfully commits an update transaction, its commit timestamp is returned. $LatestTime$ maintained by a client is updated to the value returned by the last transaction completed. When a client starts a new transaction, it sends $LatestTime$ to the originating partition for that transaction. If $LatestTime$ is greater than the current clock value at that partition, it is blocked until the clock proceeds past $LatestTime$. Otherwise, it starts immediately.

Recovery. We employ traditional recovery techniques to recover a partition in Clock-SI. Each partition maintains a write-ahead log (WAL) containing the transaction update records (as redo records), commit records, as well as 2PC prepare records containing the identity of the coordinator. In addition, the partition uses checkpointing to reduce the recovery time. Taking a checkpoint is the same as reading a full snapshot of the partition state. If a partition crashes, it recovers from the latest complete checkpoint, replays the log, and determines the outcome of prepared but not terminated transactions from their coordinators.

IV. Analytical Model

In this section, we assess the performance properties of Clock-SI analytically. Our objective is to reason about how various factors impact the performance of Clock-SI. We show the following: (1) With normal database configurations, the delay probability of a transaction is small and the delay duration is short. (2) Taking an older snapshot reduces the delay probability and duration, but slightly increases the abort rate of update transactions.

We derive formulas for transaction delay probability, delay duration, and abort rate. We verify the model predictions in Section V-E using a distributed key-value store. Readers who are not interested in the mathematical derivations of the analytical model may skip this section.

A. Model Parameters

Our model is based on prior work that predicts the probability of conflicts in centralized [21] and replicated databases [22], [18]. We consider a partitioned data store that runs Clock-SI. The data store has a fixed set of items. The total number of items is *DBSize*. We assume all partitions have the same number of items. There are two types of transactions: read transactions and update transactions. Each read transaction reads R items and takes L_r time units to finish. Each update transaction updates W items and takes L_u time units to finish. The data store processes TPS_u update transactions per unit time.

The *committing window* of update transaction T_i, denoted as CW_i, is the time interval during which T_i is in the committing state. On average, an update transaction takes CW time units to persist its writeset to stable storage synchronously. We denote RRD as the message request-reply delay, which includes one round-trip network delay, data transfer time, and message processing time. We denote S as the time to synchronously commit an update transaction to stable storage. Δ, the snapshot age, is a non-negative value subtracted from the snapshot timestamp read from the physical clock to obtain an older snapshot, as shown in Algorithm 1.

B. Delay due to Pending Commit

A transaction might be delayed when it reads an item updated by another transaction being committed to stable storage.

For a read transaction T_i, it takes its snapshot at time ST_i. Assume another update transaction T_j obtains its commit timestamp at time CT_j. With Clock-SI, at time CT_j, T_j still needs CW time to synchronously persist its writeset to stable storage. Assume T_j obtains its commit timestamp before T_i takes its snapshot, i.e., $CT_j < ST_i$. If $ST_i - CT_j < CW$, when T_i reads an item updated by T_j, it is delayed until T_j completes the commit to stable storage. Other update transactions T_k, $ST_i - CT_k > CW$, do not delay T_i, because at the time T_i reads items updated by T_k, T_k must have completed. Taking a Δ old snapshot is equivalent to shortening the committing window of update transactions to $CW - \Delta$. In this case, T_i is delayed by T_j if $ST_i - CT_j < CW - \Delta$. If $\Delta > CW$, $CW - \Delta$ effectively becomes zero.

During $CW - \Delta$, $(CW - \Delta) * TPS_u$ transactions update $(CW - \Delta) * TPS_u * W$ data items. The probability that T_i reads any particular item in the database is $R/DBSize$. If $CW - \Delta \geqslant L_r$, then the probability that T_i is delayed is $(CW - \Delta) * TPS_u * W * (R/DBSize)$. If $CW - \Delta < L_r$, then only reading the first $R * (CW - \Delta)/L_r$ items possibly delays T_i. The probability becomes $(CW - \Delta) * TPS_u * W * (R * (CW - \Delta)/L_r/DBSize)$.

For a transaction that only updates one partition, its committing window is $CW = S$ and its commit timestamp is available at the updated partition. For a transaction that updates multiple partitions, its commit timestamp is only available at its originating partition before 2PC completely finishes. Hence a delayed transaction may take one extra round-trip network latency to obtain a commit timestamp from the update transaction's originating partition. We use $CW = S + RRD$ as an estimation of the committing window of a distributed update transaction.

We assume all update transactions update items at multiple partitions. Combining the above analysis, the delay probability of short read transactions, i.e., $CW - \Delta \geqslant L_r$, is

$$(S + RRD - \Delta) * TPS_u * W * R/DBSize$$

The delay probability of long read transactions, i.e., $CW - \Delta < L_r$, is

$$(S + RRD - \Delta)^2 * TPS_u * W * R/(DBSize * L_r)$$

The expected delay duration of read transactions is

$$0.5 * (S + RRD - \Delta)$$

The above results show that the delay duration is bounded and normally short. Although the delay probability depends on various factors, we show that, with normal database configurations and workloads, it is low by a numerical example in Section IV-E and experiments in Section V-E. By assigning an older snapshot to a transaction, we can reduce its delay probability and shorten its delay duration.

C. Delay due to Clock Skew

Imperfect time synchronization causes clock skew. A transaction is delayed when it accesses a remote partition and the remote clock time is smaller than its snapshot timestamp. As we show in Section V, common clock synchronization protocols, such as NTP, work well in practice and the clock skew is very small. Hence, this type of delay rarely happens.

We assume the clock skew SK between each pair of clocks follows normal distribution [23] with mean μ and standard deviation δ. The delay probability when a transaction accesses a remote partition is

$$P(SK > 0.5 * RRD + \Delta) = 1 - \Phi((0.5 * RRD + \Delta - \mu)/\delta)$$

The expected delay duration is

$$\frac{\int_{0.5*RRD+\Delta}^{+\infty} x e^{-(x-\mu)^2/(2\delta^2)}\, dx}{\int_{0.5*RRD+\Delta}^{+\infty} e^{-(x-\mu)^2/(2\delta^2)}\, dx} - (0.5 * RRD + \Delta)$$

The results show that if the clock skew is greater than one-way network latency, it becomes possible that a transaction is delayed when accessing a remote partition, because the requested snapshot is not yet available when the transaction arrives. By assigning an older snapshot to a transaction, we can reduce its delay probability and shorten its delay duration.

Suppose the maximum clock skew is SK_{max} time units. Taking an older snapshot with $\Delta = max(CW, SK_{max} - 0.5 * RRD)$ eliminates almost all the delays due to either pending commits or clock skews.

D. Update Transaction Abort Probability

Assigning old snapshots to transactions reduces their delay probability and duration. However, this increases the abort rate of update transactions because the execution time of an update transaction is extended and more update transactions run concurrently with it.

We first compute the probability that a transaction T_i has to abort without adjusting the snapshot timestamp [18]. On average, the number of transactions that commit in L_u, the life time of T_i, is $TPS_u * L_u$. The number of data items updated by the these transactions is $W * TPS_u * L_u$. The probability that one particular item in the database is updated during L_u is $W * TPS_u * L_u / DBSize$. As T_i updates W items in total, the probability that T_i has conflicts with its concurrent transactions and has to abort is $W^2 * TPS_u * L_u / DBSize$. The abort probability is directly proportional to L_u, the duration of update transaction execution.

Clock-SI assigns to each transaction a snapshot that is Δ older than the latest snapshot. The transaction abort probability becomes

$$W^2 * TPS_u * (L_u + \Delta)/DBSize$$

This is the upper bound of the abort rate because assigning an older snapshot to a transaction does not extend its execution time physically. The actual transaction execution time remains unchanged. It only becomes longer *logically*.

With other parameters fixed, the longer a transaction executes, the more concurrent transactions run with it, increasing the likelihood of write-write conflicts with other transactions. Assigning an older snapshot to a transaction increases its abort probability.

E. Example

We provide a numerical example using the equations to calculate the transaction delay probability and duration due to pending commits of update transactions. We assume $DBSize = 10,000,000$, $R = 10$, $W = 10$, $TPS_u = 10,000$, $RRD = 0.2ms$, $\Delta = 0$, $CW = 8ms$ and $L_u = 32ms$. If we use mechanical hard disks to persist transaction updates, it takes a few milliseconds (e.g., 8ms) to commit an update transaction. The delay probability is 0.08% and the expected delay time is 4.1ms. On average, each transaction is delayed for 3.2μs. If a transaction takes a snapshot that is 8.2ms earlier than the clock time, i.e., $\Delta = 8.2ms$, both delay probability and time are zero, and the transaction abort probability increases by 25% (from 0.0032 to 0.004). Therefore we can almost eliminate transaction delays at the cost of a slight increase in the abort rate.

V. Experimental Evaluation

We evaluate the performance benefits of Clock-SI using a micro-benchmark and an application-level benchmark with a partitioned key-value store in both LAN and WAN environments, and show the following: (1) Compared with conventional SI, Clock-SI reduces transaction response time and increases throughput. (2) Selecting a slightly older snapshot reduces the delay probability and duration of transactions. Furthermore, we verify the predictions of our analytical model using experimental measurements. Notice that we focus only on assessing the performance benefits of Clock-SI, rather than the improvement in availability stemming from avoiding a single point of failure.

A. Implementation and Setup

We build a partitioned multiversion key-value store in `C++`. It supports Clock-SI as well as conventional SI as implemented in other systems [3]. With conventional SI, a transaction communicates with a centralized timestamp authority running on a separate server to retrieve timestamps. A transaction needs one round of messages to obtain the snapshot timestamp. An update transaction needs another round of messages to obtain the commit timestamp.

The data set is partitioned among a group of servers. Servers have standard hardware clocks synchronized using NTP running in peer mode [15]. A key is assigned to a

partition based on its hash. The default size of a key and its value are 8 and 64 bytes, respectively. We keep all key-value pairs in a group of hash tables in main memory. A key points to a linked list that contains all the versions of the corresponding value. The transaction commit log resides on the hard disk. The system performs group commit to write multiple transaction commit records in one stable disk write.

We conduct experiments in both LAN and WAN environments. For LAN experiments, we deploy our key-value store in a local cluster. Servers in our local cluster run Linux 3.2.0. Each server has two Intel Xeon processors with 4GB DDR2 memory. The transaction log resides on a 7200rpm 160GB SATA disk. We disable the disk cache so that synchronous writes reach the disk media. The average time of a synchronous disk write is 6.7ms. The system has eight partitions. Each partition runs in one server and manages one million keys in main memory. All machines are connected to a single Gigabit Ethernet switch. The average round-trip network latency is 0.14 milliseconds. For WAN experiments, we deploy our system on Amazon EC2 using medium instances (size M1) at multiple data centers.

B. Micro-benchmarks

We implement a benchmark tool based on the Yahoo! Cloud Serving Benchmark (YCSB) [13], which is designed to benchmark key-value stores. We extend YCSB to support transactions since the original YCSB does not support transactions.

Latency. Clock-SI takes shorter time to run a transaction because it does not communicate with a timestamp authority for transaction snapshot and commit timestamps. Figure 3 shows the latency of read-only transactions in our local cluster. Both the clients and servers are connected to the same switch. Each transaction reads eight items at each partition, with keys chosen uniformly randomly. We vary the number of partitions accessed by a transaction from one to three. Compared with conventional SI, Clock-SI saves approximately one round-trip latency. For a single-partition read-only transaction this amounts to about 50% latency savings. We also see consistent savings of one round-trip latency for the other two cases.

We run the same experiment in a WAN environment on Amazon EC2 and measure transaction latency. We place three data partitions at three data centers in different geographical locations: US West, US East and Europe. The timestamp server is co-located with the partition in US West. A client always chooses the nearest partition as the originating partition of its transaction.

First, we run clients at our local cluster in Europe and the number of partitions accessed by a transaction varies from one to three. Figure 4 shows the measured latency. Compared with conventional SI, Clock-SI reduces the latency of each transaction by about 160 milliseconds since it does not contact the remote timestamp server. Next, for transactions issued by the clients near US West and served by the partition co-located with the timestamp server, the saved latency by Clock-SI becomes sub-milliseconds and the latencies are similar to those

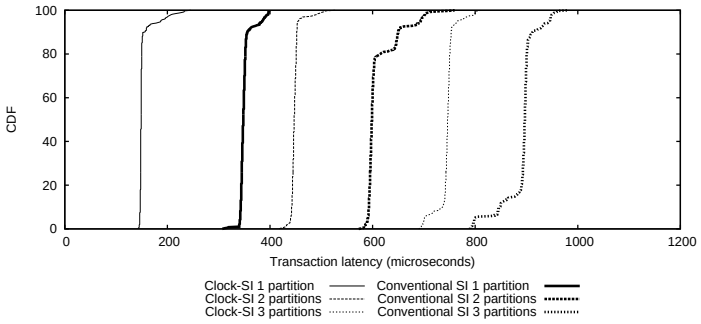

Fig. 3. Latency distribution of read-only transactions in a LAN.

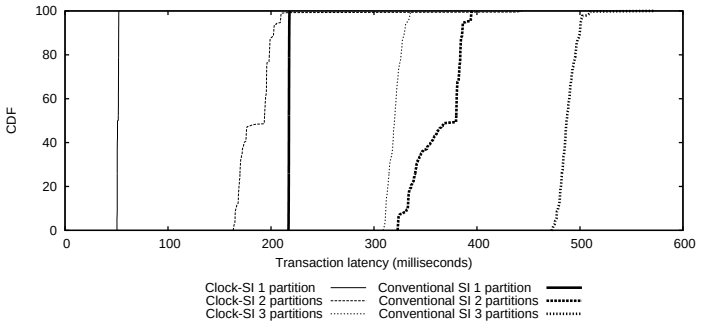

Fig. 4. Latency distribution of read-only transactions in a WAN. Partitions are in data centers in Europe, US West and US East. Clients are in our local cluster in Europe.

in Figure 3. Last, for the clients near US East, Clock-SI still reduces their transaction latency by one round-trip between two data centers, which is about 170 milliseconds.

For update transactions, the latency is reduced similarly by two network round trips in both LAN and WAN environments.

Throughput. Next we compare the throughput of Clock-SI with conventional SI. We run a large number of clients to saturate the servers that host the key-value data partitions. Figure 5 shows the throughput of read-only and update-only single-partition transactions. The number of partitions serving client requests varies from one to eight. Each transaction reads or updates eight items randomly chosen at each partition.

We first analyze read-only transactions. Below five partitions, the throughput of Clock-SI is about twice of conventional SI. The cost of a read-only transaction stems mainly from sending and receiving messages. As Clock-SI does not contact the timestamp authority to obtain snapshot timestamps, it sends and receives one message for each transaction, while conventional SI sends and receives two messages. Beyond five partitions, the throughput gap becomes larger: The throughput of Clock-SI increases linearly as the number of partitions increases. For conventional SI, the throughput levels off at around 64k transactions/second. The timestamp authority becomes the bottleneck because it can only assign about 64k timestamps/second.

Update transactions show similar results in Figure 5. The throughput of update transactions is lower than that of read-only transactions, because an update transaction does more work, including creating new versions of items, updating version metadata, and performing I/O to make updates durable on the disk. Update transactions require two timestamps from the

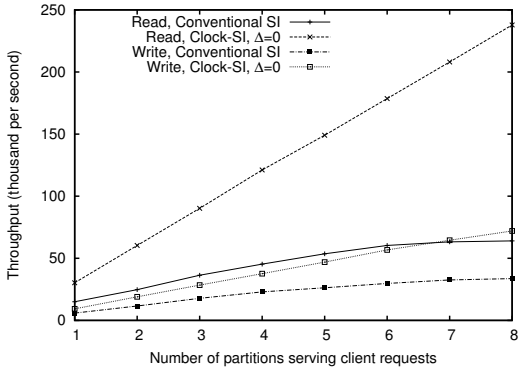

Fig. 5. Throughput comparison for single-partition read-only and update-only transactions. Each transaction reads/updates eight items.

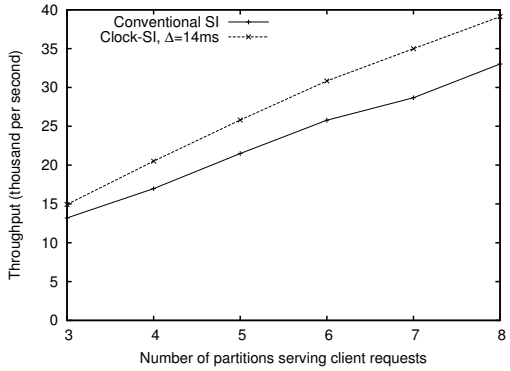

Fig. 6. Throughput of Twitter Feed-Following application. 90% read-tweets and 10% post-tweets.

timestamp authority. Given its limit of 64k timestamps/second, the timestamp authority sustains only 32k update transactions/second. The timestamp authority again becomes the bottleneck.

Batching timestamp requests from the data partitions to the timestamp authority can improve the throughput of conventional SI. However, message batching comes with the cost of increased latency. In addition, in a system with large number of data partitions, even with message batching, the centralized timestamp authority can still become a bottleneck under heavy workloads.

The results of our micro-benchmarks show that Clock-SI has better performance than conventional SI in both LAN and WAN environments as it does less work per transaction, improving both latency and throughput. We show the results of transactions containing both read and update operations for an application benchmark in the next section.

C. Twitter Feed-Following Benchmark

We build a Twitter-like social networking application on top of the distributed key-value store. The feed-following application supports *read-tweet* and *post-tweet* transactions on a social graph. The transactions guarantee that a user always reads consistent data. Each user in this application has a key to store the friend list, a key for the total number of tweets, and one key for each tweet. There is a trade-off between pushing tweets to the followers and pulling tweets from the followees [24]. We choose the push model, which optimizes for the more common read transactions.

We model 800,000 users. The followers of a user are located at three partitions. On average, each user follows 20 other users. The users accessed by the read-tweet and post-tweet transactions are chosen uniformly randomly. A post-tweet transaction pushes a tweet of 140 characters to the followers of a user at three different partitions. A read-tweet transaction retrieves the 20 latest tweets from the followees, which accesses one partition. The workload includes 90% read-tweet transactions and 10% post-tweet transactions, as used in prior work [14].

Figure 6 shows the throughput of Clock-SI and conventional SI. With eight partitions Clock-SI supports more than 38k transactions/second, while conventional SI supports 33k transactions/second. The average transaction latency of Clock-SI is also lower than that of conventional SI by one round-trip latency for the read-tweet transactions and two round-trip latency for the post-tweet transactions (figure not shown).

Since the transactions access a reasonably large data set uniformly randomly and clocks are well synchronized by NTP, transaction delays rarely occur and do not affect the overall performance. We discuss transaction delays with carefully designed workloads further below.

D. Effects of Taking Older Snapshots

In previous experiments, the delay probabilities are very small because there are few "conflicts" between reads and pending commits. Here we change the workload to create more conflicts using a small data set. We show how choosing a proper value for Δ, the snapshot age, in Clock-SI reduces the delay probability and duration. We run a group of clients and each client issues update transactions that modify ten items. Another group of clients issue transactions that read ten of the items being updated with varying probability. We vary Δ and measure transaction throughput and latency.

Figure 7 shows the throughput of read-only transactions with different Δ values against the probability of reading hot-spot items. Figure 8 shows the corresponding latency. As Δ becomes larger, the throughput increases and the latency decreases. With Δ greater than the duration of a commit, reads are not delayed by updates at all (for curves with $\Delta = 14ms$ and $\Delta = 21ms$). With smaller Δ values, the probability that reads are delayed increases when the probability that a transaction reads the hot-spot items increases. As a result, the transaction latency increases and the throughput drops. Therefore, choosing a proper snapshot age in Clock-SI effectively reduces the probability of transaction delays.

E. Model Verification

We run experiments to verify our analytical model and the effects of using different Δ values in Clock-SI. Each transaction both reads and writes a fixed number of data items at one partition. We check whether the delay probability and duration change as the model in Section IV predicts when we change the data set size and snapshot age.

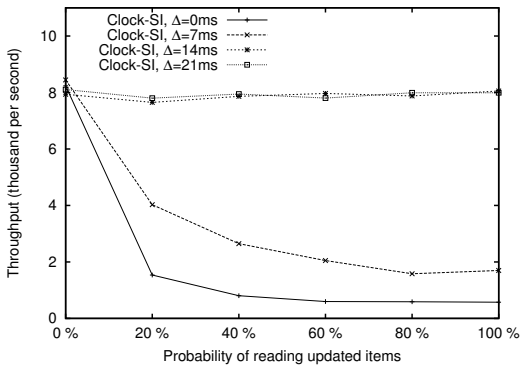

Fig. 7. Read transaction throughput with write hot spots.

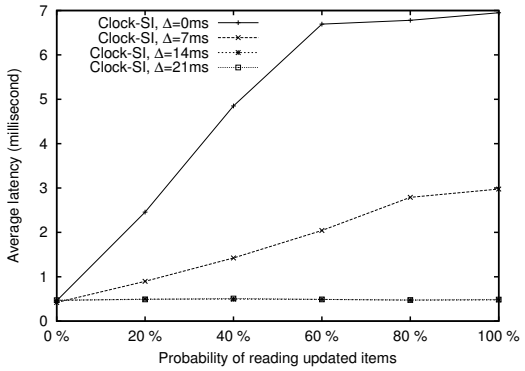

Fig. 8. Read transaction latency with write hot spots.

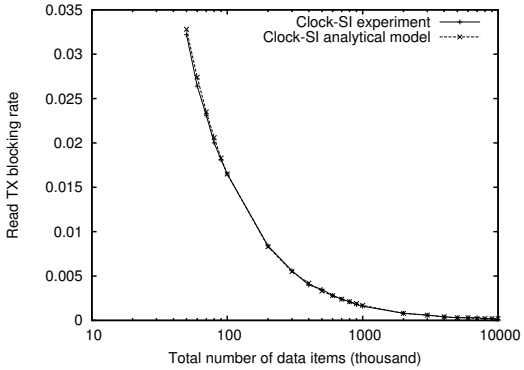

Fig. 9. Transaction delay rate when $\Delta = 0$ while varying the total number of data items.

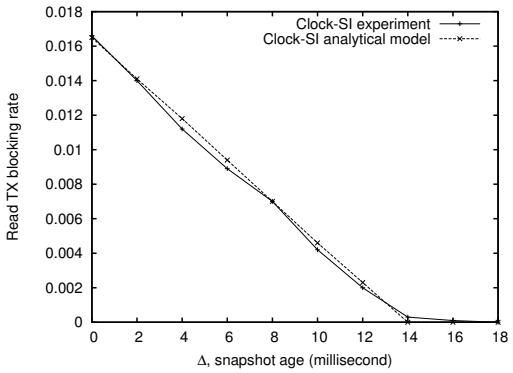

Fig. 10. Transaction delay rate in a small data set while varying Δ, the snapshot age.

Figure 9 shows that the delay probability decreases with the size of the data set. The numbers produced by the analytical model follow the same pattern as the experimental results. As each transaction accesses items uniformly randomly, the larger the data set, the less likely that a transaction reads an item updated by another committing transaction.

Next we show that taking older snapshots reduces both the transaction delay probability and duration. We choose a small data set with 50,000 items to make the delays happen more often. Figure 10 and 11 demonstrate the effects of choosing different snapshot ages. The older a snapshot, the lower the delay probability and the shorter the delay duration. Figure 12 shows how the transaction abort rate changes. As the analytical model predicts, the transaction abort rate increases as the age of the snapshot increases.

F. NTP Precision

With Clock-SI, a transaction might be delayed when accessing a remote partition if the remote clock time is smaller than the snapshot timestamp of the transaction. The occurrence of this type of delay indicates that the clock skew is greater than one-way network latency. In all the experiments, we observe that most of the transaction delays are due to the pending commit of update transactions. Only very few delays are due to clock skew.

We measure the synchronization precision of NTP indirectly on our local cluster since it is difficult to measure the time difference of two clocks located at two different servers directly. We record the clock time, t_1, at server s_1 and immediately send a short message containing t_1 to another server s_2 over the network. After the message arrives at s_2, we record its clock time, t_2. $t_2 - t_1$ is the skew between the two clocks at s_2 and s_1 plus one-way network latency. If $t_2 - t_1 > 0$, Clock-SI does not delay transactions that originate from s_1 and access data items at s_2. Figure 13 shows the distribution of the clock skew between two clocks plus one-way network latency measured every 30 seconds in six weeks. A negative value on the x axis indicates the possibility of delaying a transaction when accessing a remote partition. As we see from the figure, the delay probability due to clock skew is very low and the

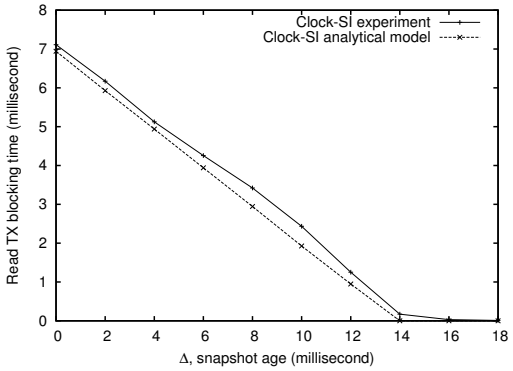

Fig. 11. Transaction delay time in a small data set while varying Δ, the snapshot age.

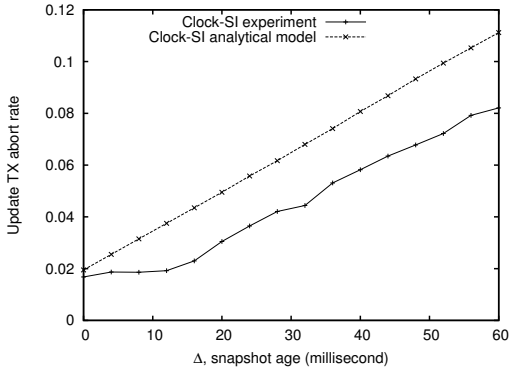

Fig. 12. Transaction abort rate in a small data set while varying Δ, the snapshot age.

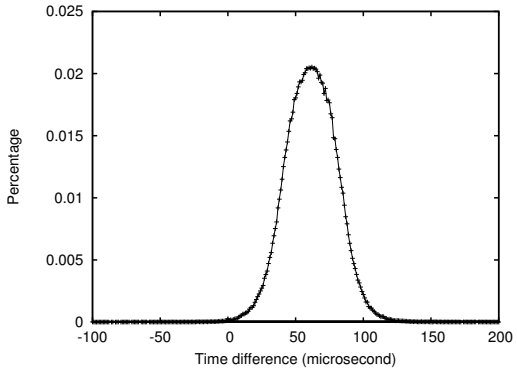

Fig. 13. Distribution of the clock skew between two clocks plus one-way network latency in LAN.

delay duration is very short.

VI. Related Work

SI in distributed systems. A number of large-scale distributed data stores use SI to support distributed transactions. They all rely on a centralized service for timestamp management. Percolator [3] adds distributed SI transactions to Bigtable [25]. Zhang and Sterck [4] implement SI on top of HBase in a system that stores all the transaction metadata in a number of global tables. An implementation of write snapshot isolation (WSI) using a centralized transaction certifier is given in [9]. Clock-SI uses physical clocks rather than a centralized authority for timestamp management, with the attendant benefits shown in this paper.

Physical clocks in distributed systems. Liskov provides a survey of the use of loosely synchronized clocks in distributed systems [16]. Schneider uses loosely synchronized clocks to order operations for state machine replication [10] such that all replicas execute operations in the same serial order according to the physical time when the operations are initialized at the replicas. In contrast, Clock-SI uses physical clocks for a different purpose, providing consistent snapshots for transactions accessing a partitioned data store.

The Thor project explores the use of loosely synchronized clocks for distributed concurrency control [11], [12], [26]. AOCC [11] assigns to committed transactions unique commit timestamps from physical clocks. Transactions running under AOCC may, however, read inconsistent states of the database. Therefore, read-only transactions need to be validated on commit and therefore may need to abort, an undesirable situation which does not happen in Clock-SI.

An extension to AOCC lets running transactions read *lazily consistent states* [12] according to a dependency relation providing *lazy consistency* (LC). LC is weaker than SI [17]. Some read histories allowed under LC are forbidden by SI. For example, assume two items x_0 and y_0. Transaction T_1 writes x_1 and commits. Then transaction T_2 writes y_1 and commits. Next, transaction T_3 reads the two items. Under LC, T_3 is allowed to read x_0 and y_1, which is not serializable and also not allowed under SI. Therefore, even with AOCC+LC read-only transactions need to be validated and may have to abort.

Both AOCC and AOCC+LC do not provide consistent snapshots for running transactions. In comparison, transactions in Clock-SI always receive consistent snapshots and read-only transactions do not abort.

Spanner [27] implements serializability in a geographically replicated and partitioned data store. It provides external consistency based on synchronized clocks with bounded uncertainty, called TrueTime, requiring access to GPS and atomic clocks. Spanner uses conventional two-phase locking for update transactions to provide serializability. In addition, transactions can be annotated as read-only and executed according to SI. In comparison, Clock-SI relies solely on physical time to implement SI. Spanner's provision of external consistency requires high-precision clocks, and its correctness depends on clock synchrony. In contrast, Clock-SI uses conventional physical clocks available on today's commodity servers, and its correctness does not depend on clock synchrony.

Granola [28] runs single-partition and independent multi-partition transactions serially at each partition to remove the cost of concurrency control. Such a transaction obtains a timestamp before execution, and transactions execute serially in timestamp order. Coordinated multi-partition transactions use traditional concurrency control and commit protocols. To increase concurrency on multicore servers, Granola partitions the database among CPU cores. This increases the cost of transaction execution, because transactions that access multiple partitions on the same node need distributed coordination. In contrast, Clock-SI runs all transactions concurrently and does not require partitioning the data set among CPU cores.

Relaxing SI in distributed systems. Prior work proposes relaxing the total order property of SI to achieve better performance in partitioned and replicated systems. Walter [14] is a transactional geo-replicated key-value store that uses parallel snapshot isolation (PSI), which orders transactions only within a site and tracks causally dependent transactions using a vector clock at each site, leaving independent transactions unordered among sites. Non-monotonic snapshot isolation (NMSI) [29], [30] provides non-monotonic snapshots. NMSI does not explicitly order transactions but uses *dependency vectors* to track causally dependent transactions. NMSI improves PSI by supporting *genuine* partial replication. The relaxations of SI

may fail to provide application developers with the familiar isolation levels and requires extra effort to guarantee that transactions read consistent and monotonic snapshots as in SI. In contrast, Clock-SI provides a complete implementation of SI, including a total order on transaction commits and a guarantee that transactions read consistent and monotonic snapshots across partitions.

Relaxing freshness. Some systems relax freshness to improve performance at the cost of serving stale data. Relaxed currency models [31], [32], [33] allow each transaction to have a freshness constraint. Continuous consistency [34] bounds staleness using a real-time vector. These systems do not use physical clocks to assign transaction snapshot and commit timestamps. Clock-SI provides each transaction with either the latest snapshot or a slightly older snapshot (tunable per transaction) to reduce the delay probability and duration of transactions. In both cases, all the properties of SI are maintained.

Generalized snapshot isolation (GSI) [18] generalizes SI to replicated databases. It uses older snapshots to avoid the delay of waiting for committed updates to be propagated from the certifier to the replicas. In contrast, Clock-SI targets concurrency control for partitioned data stores using physical time. It allows assigning older snapshots to reduce the delay probability and duration of transactions.

VII. Conclusions

Clock-SI is a fully distributed implementation of SI for partitioned data stores. The novelty is the provision of consistent snapshots using loosely synchronized clocks. Clock-SI uses physical clocks for assigning snapshot and commit timestamps. It improves over existing systems that use a centralized timestamp authority, by eliminating a single point of failure and a potential performance bottleneck. Moreover, Clock-SI avoids the round-trip latency between the partitions and the timestamp authority, showing better response times for both LAN and WAN environments.

Acknowledgments

We thank Rachid Guerraoui, Anne-Marie Kermarrec, Fernando Pedone, Masoud Saeida Ardekani, Marc Shapiro, and the anonymous reviewers for their feedback, comments and discussions which improved this paper.

References

[1] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil, "A critique of ansi sql isolation levels," in *SIGMOD 1995*.

[2] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency control and recovery in database systems*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1986.

[3] D. Peng and F. Dabek, "Large-scale incremental processing using distributed transactions and notifications," in *OSDI 2010*.

[4] C. Zhang and H. De Sterck, "Supporting multi-row distributed transactions with global snapshot isolation using bare-bones hbase," in *GRID 2010*.

[5] F. Junqueira, B. Reed, and M. Yabandeh, "Lock-free transactional support for large-scale storage systems," in *HotDep 2011*.

[6] S. Elnikety, S. Dropsho, and F. Pedone, "Tashkent: uniting durability with transaction ordering for high-performance scalable database replication," in *EuroSys 2006*.

[7] K. Daudjee and K. Salem, "Lazy database replication with snapshot isolation," in *VLDB 2006*.

[8] S. Wu and B. Kemme, "Postgres-r (si): Combining replica control with concurrency control based on snapshot isolation," in *ICDE 2005*. IEEE.

[9] D. Ferro and M. Yabandeh, "A critique of snapshot isolation," in *Eurosys 2012*.

[10] F. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys (CSUR)*, vol. 22, no. 4, pp. 299–319, 1990.

[11] A. Adya, R. Gruber, B. Liskov, and U. Maheshwari, "Efficient optimistic concurrency control using loosely synchronized clocks," in *SIGMOD 1995*.

[12] A. Adya and B. Liskov, "Lazy consistency using loosely synchronized clocks," in *PODC 1997*.

[13] B. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *SOCC 2010*.

[14] Y. Sovran, R. Power, M. Aguilera, and J. Li, "Transactional storage for geo-replicated systems," in *SOSP 2011*.

[15] "The network time protocol," http://www.ntp.org, 2012.

[16] B. Liskov, "Practical uses of synchronized clocks in distributed systems," *Distributed Computing*, vol. 6, no. 4, pp. 211–219, 1993.

[17] A. Adya, "Weak consistency: a generalized theory and optimistic implementations for distributed transactions," Ph.D. dissertation, Massachusetts Institute of Technology, 1999.

[18] S. Elnikety, F. Pedone, and W. Zwaenepoel, "Database replication using generalized snapshot isolation," in *SRDS 2005*.

[19] K. Daudjee and K. Salem, "Lazy database replication with ordering guarantees," in *ICDE 2004*.

[20] K. Krikellas, S. Elnikety, Z. Vagena, and O. Hodson, "Strongly consistent replication for a bargain," in *ICDE 2010*.

[21] J. Gray and A. Reuter, *Transaction processing*. Kaufmann, 1993.

[22] J. Gray, P. Helland, P. O'Neil, and D. Shasha, "The dangers of replication and a solution," in *SIGMOD 1996*.

[23] J. Elson, L. Girod, and D. Estrin, "Fine-grained network time synchronization using reference broadcasts," *ACM SIGOPS Operating Systems Review*, vol. 36, 2002.

[24] A. Silberstein, J. Terrace, B. Cooper, and R. Ramakrishnan, "Feeding frenzy: selectively materializing users' event feeds," in *SIGMOD 2010*.

[25] F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, p. 4, 2008.

[26] B. Liskov, M. Castro, L. Shrira, and A. Adya, "Providing persistent objects in distributed systems," *ECOOP99Object-Oriented Programming*, pp. 667–667, 1999.

[27] J. Corbett, J. Dean *et al.*, "Spanner: Googles globally-distributed database," *OSDI*, 2012.

[28] J. Cowling and B. Liskov, "Granola: low-overhead distributed transaction coordination," in *USENIX ATC 2012*.

[29] M. Saeida Ardekani, P. Sutra, N. Preguiça, and M. Shapiro, "Non-Monotonic Snapshot Isolation," Tech. Rep. RR-7805, Jun. 2013.

[30] M. Saeida Ardekani, P. Sutra, and M. Shapiro, "Non-monotonic snapshot isolation: Scalable and strong consistency for geo-replicated transactionalsystems," in *SRDS*, 2013.

[31] H. Guo, P. Larson, P. Ramakrishnan, and J. Goldstein, "Relaxed currency and consistency: how to say good enough in sql," in *SIGMOD 2004*.

[32] H. Guo, P. Larson, and Ramakrishnan, "Caching with good enough currency, consistency and completeness," in *VLDB 2005*.

[33] P. Bernstein, A. Fekete, H. Guo, R. Ramakrishnan, and P. Tamma, "Relaxed-currency serializability for middle-tier caching and replication," in *SIGMOD 2006*.

[34] H. Yu and A. Vahdat, "Design and evaluation of a continuous consistency model for replicated services," in *OSDI 2000*.